CMPSC 465 Spring 2024 — Data Structures & Algorithms — Mehrdad Mahdavi and David Koslicki — HW 3

1. (20 pts.) Problem 1

(a) After Build-Heap: $[0,5,1,7,6,4,3,10,8,12]$
After 1st iteration of the loop: $[1,5,3,7,6,4,12,10,8,0]$
After 2nd iteration of the loop: $[3,5,4,7,6,8,12,10,1,0]$
After 3rd iteration of the loop: $[4,5,8,7,6,10,12,3,1,0]$

(b) Proof: Assume the element at position $\lfloor \frac{n}{2} \rfloor$ is an internal node, not a leaf. Then, its left child should be at position $2 \cdot \lfloor \frac{n}{2} \rfloor + 1 \geq 2 \cdot \frac{n}{2} - 1 + 1 = n > n-1$, which is the index of the last element in heap. Therefore, there is a contradiction. Thus, the element at position $\lfloor \frac{n}{2} \rfloor$ can't have any children, meaning it must be a leaf. The same reasoning can be applied to other positions in the statement since they are even larger than $\lfloor \frac{n}{2} \rfloor$. So, the nodes at positions $\lfloor \frac{n}{2} \rfloor, \ldots, n-1$ must be the leaves.

(c) Consider an array $[2,3,4,1]$. If we increase i from 0 to $\lfloor \frac{n}{2} \rfloor - 1$, there is no swap in the first iteration. In the second iteration, 3 swaps with 1. Then, the loop and the Build-Heap procedure are finished with the array $[2,1,4,3]$ which doesn't satisfy the min heap property since 2 is greater than its left child 1. So, increasing i from 0 to $\lfloor \frac{n}{2} \rfloor - 1$ can't guarantee that the array we end up obtaining from Build-Heap has the min heap property.

2. (15 pts.) Problem 2

We keep $M+1$ counters, one for each of the possible values of the array elements. We can use these counters to compute the number of elements of each value by a single $O(n)$-time pass through the array. Then, we can obtain a sorted version of x by filling a new array with the prescribed numbers of elements of each value, looping through the values in ascending order. This algorithm is called Counting Sort. Notice that the $\Omega(n \log n)$ bound does not apply in this case, as this algorithm is not comparison-based.

3. (15 pts.) Problem 3

(a) We get the recurrence $T(n) = T(n/2) + \Theta(1)$ for the running time of Binary Search. Applying the master theorem with $a = 1$, $b = 2$, $d = 0$ and $\log_b a = 0$, we note that we are in the second case, and so $\Theta(n^{\log_2 1} \log(n)) = \Theta(\log(n))$

(b) For Ternary Search the recurrence is $T(n) = T(n/3) + \Theta(1)$. Applying the master theorem with $a = 1$, $b = 3$, and $d = 0$, we have that $\log_3 1 = 0 = d$. So, we are in the second case, and we have that $\Theta(n^{\log_3 1} \log(n)) = \Theta(\log(n))$

4. (20 pts.) Problem 4

Let $m = \lfloor \frac{n}{2} \rfloor$, first we divide the input sequence in two halves ($A_1 = a_1, \ldots, a_m$ and $A_2 = a_{m+1}, \ldots, a_n$). Also, let $T(n)$ be the time it takes to sort the input sequence of n numbers and to count the number of inversions. First, we recursively sort, and count each half, which takes $2T(\frac{n}{2})$. Then, we should merge two halves, and count the number of inversions between the two halves. We can do this as follows. Remember that the two halves are A_1 and A_2, and let B_1, and B_2, be the output of recursive call on A_1 and A_2. Let i be the pointer to array B_1, and j be the pointer to array B_2. Also initialize i and j to zero (for example $i = 0$ points to first element of array B_1). (Merge part:) Compare the first elements of B_1, and B_2, and increase the pointer of the smaller element by one, and put it in output array S.(Count part:) Also, whenever we increase the B_2

pointer by one, we increase the total number of inversions by $\texttt{length}(B_1) - i$, since whenever we increase j (or B_2 pointer) by one, $B_2[j]$ should be less than $B_1[i : \frac{n}{2}]$, which makes $\texttt{length}(B_1) - i$ inversions. To get more details, You can see the pseudocode in algorithm 1.

Algorithm 1 sort, and count

Input: an array $A = [a_1, a_2, \ldots, a_n]$, and its length n
function SORT-COUNT(A, n)
  if $n == 1$ **then** return $A[0], 0$
  end if
  $A_1 \leftarrow [a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}]$, $A_2 \leftarrow [a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n]$
  $B_1, cnt_1 \leftarrow \texttt{sort-count}(A_1, \lfloor \frac{n}{2} \rfloor)$
  $B_2, cnt_2 \leftarrow \texttt{sort-count}(A_2, \lceil \frac{n}{2} \rceil)$
  $i \leftarrow 0$
  $j \leftarrow 0$
  $S \leftarrow$ an array of length n
  $k \leftarrow 0$
  $cnt \leftarrow 0$
  while $i < \texttt{length}(B_1)$ and $j < \texttt{length}(B_2)$ **do**
    if $B_2[j] \geq B_1[i]$ **then**
      $S[k] = B_1[i]$, $i \leftarrow i + 1$
    else
      $S[k] = B_2[j]$, $j \leftarrow j + 1$, $cnt \leftarrow cnt + \texttt{length}(B_1) - i$
    end if
    $k \leftarrow k + 1$
  end while
  if $i = \texttt{length}(B_1)$ **then**
    $S[k : n-1] = B_2[j : n-1]$
  end if
  if $j = \texttt{length}(B_2)$ **then**
    $S[k : n-1] = B_1[i : n-1]$
  end if
  return S, $cnt_1 + cnt_2 + cnt$
end function

Now since the merge part takes at most n iteration of while loop, we can say the running time for merge is $O(n)$. So, the recursive relation would be:

$$T(n) = 2T(\frac{n}{2}) + O(n) \tag{1}$$

If we use the master theorem for the above recursive relation, we have $T(n) = O(n \log n)$

5. (30 pts.) Problem 5

Maintain 2 heaps. Using induction, assume that there is (a) a max heap for the lower (in value) half of the elements, and (b) a min heap for the upper half of the elements. Given a new element, check the root of the larger (in number of elements) heap.

1. If the larger heap is the max heap, then, if the element is smaller, insert the element into the max heap, delete the root, insert the root into the min heap. If the element is bigger, insert it into the min heap. After 1, (a) and (b) are still satisfied.

2. If the larger heap is the min heap, then, if the element is bigger, insert the element into the min heap, delete the root, insert the root into the max heap. If the element is smaller, insert it into the max heap. After 2, (a) and (b) are still satisfied.

3. If both heaps are the same size, choose the root of the min heap arbitrarily. If the element is bigger, insert the element into the min heap. If the element is smaller, insert it into the max heap. After 3, (a) and (b) are still satisfied.

4. If both heaps are the same size, the median is the average of the roots. If one heap is larger, the median is the root of the larger heap.

Following 1, 2, 3, and the properties of heaps, (a) and (b) will always be satisfied, so 4 always returns the median.

Note that in the base case of the induction of two empty heaps, the element can be inserted into a heap arbitrarily.

There are a constant number of operations taking $O(\log n)$ for each number, so the final runtime is as such.

6. (0 pts.) Acknowledgments

(a) I did not work in a group.
(b) I did not consult with anyone other than my group members.
(c) I did not consult any non-class materials.

Rubric:

Problem 1, total points 20

(a) 6 points
3 points: provide correct resulting array for Build-Heap
1 point: provide correct resulting array for the 1st iteration
1 point: provide correct resulting array for the 2nd iteration
1 point: provide correct resulting array for the 3rd iteration

(b) 7 points
4 points: explain the child issue
3 points: explain the child issue is applicable to all the elements in the specified positions

(c) 7 points
3 points: provide a valid counterexample for increasing i from 0 to $\lfloor \frac{n}{2} \rfloor - 1$
4 points: explain why this increasing i from 0 to $\lfloor \frac{n}{2} \rfloor - 1$ fails on this counterexample

Problem 2, 15 pts

- 8 points for proof of time complexity
- 7 points for the justification of why $\Omega(n \log n)$ doesn't apply to the case where M is small and the time is linear time.

Problem 3,15 pts

- 4 points for recurrence $T(n) = T(n/2) + \Theta(1)$, 4 points for running time $\Theta(n)$
- 3.5 points for recurrence $T(n) = T(n/3) + \Theta(1)$, 3.5 points for running time $\Theta(n)$

Problem 4, 20 pts.

The algorithm includes two recursive call, and then a merge and count part.
Designing a correct merge and count procedure: 5
The correctness of algorithm: 5
Deriving the recurrence relation: 5
Showing the run-time from the recurrence relation: 5

Problem 5, 30 pts

Explaining running time: 5
Explaining correctness: 5
Correct algorithm: 20